Lucrare de Diplomă

Towards the Unification of Static Analysis Mechanisms for Object-Oriented Systems

Cristina Sofonea

Universitatea *"Politehnica"* Timişoara
Facultatea de Automatică şi Calculatoare
Departamentul de Calculatoare şi Inginerie Software

Conducător ştiinţific:
ing. Radu Marinescu

Iunie, 2002

Contents

Chapter 1

Introduction

1.1 Motivation

When we have in front of us the source of a program, besides the functional accuracy we are also interested in the quality of the design or of the implementation as it is reflected in the source code. This is what code analysis is all about. If the source code is as small as a few tens of lines, we can examine the code manually; but as the source code becomes larger we want to find out a lot of additional things like the impact of change a class functionality on the other classes.

Static source code analysis is the process by which software developers check their code for problems and inconsistencies before compiling. Development organizations have two basic alternatives for conducting static source code analysis: manual code check or automatic static source code analysis. Automatic source code static analysis addresses many difficult problems faced by the software industry today [9]:

- Software is hard to managed
  - Software systems have become larger and more complex
  - Software system often needs to be modified to meet new requirements or to add new features. Making change to a software system, however, is risky because it may introduce inconsistencies to other modules of the code
- Lack of code inspection tools
  - Software written in multiple files in textual format is hard to read and understand

  - All software systems are subject to bugs. Code inspection is known to help find about half the bugs, but it is very difficult to perform because the code is not traceable

- Lack of easily-accessible documentation
  - Documenting code is a time-consuming and difficult task, especially since documentations need to be updated with each round of code modification
  - For software systems that do have documentations at physic design level and program-level, those documents are often out of date and do not correspond to the code
  - In many systems the documents were written on paper, and not accessible on-line

- Lack of good documentation tools
  - Some tools can generate documents from users' source code and show them on the computer screen, but the output is static and hard to read through
  - Often the database of a document tools is not incremental, so that when the user changes a file all other files in the same project must be re-analyzed again to get updated documents
  - Design documents and diagrams for the most systems are drawn using graphic editors inefficiently, and hard to modify

- Lack of good system structure analysis tools
  - Most of the available system structure analysis tools are function-based, they cannot be used to accurately analyze the structure of an object-oriented system

- Lack of automatic quality assurance measurement
  - It is necessary to establish quality standards for a company. But without an automatic tool it is difficult to set up and use quality standards in practices, including metric selection, metric boundary value setting, and metric boundary value modification
  - Collecting the quality data of a software system is difficult. In many companies, this task is done manually, takes a lot of time. Often the collected quality information is out of date and incorrect

- Most quality measurement tools available on the market are function-based. It is virtually impossible to use those function-based quality measurement tools to measure an object-oriented software system precisely. For example, for unit testing an object-oriented system, class is the most important unit. But a function-based quality measurement tool will only report the results by function rather than by class
- Measuring the quality of an object-oriented software system is much more difficult than measuring a traditional software system. For instance, the complexity and test coverage of a class need to be determined indirectly; a class may be derived from many base classes which also need to be counted
- There is inadequate class-based metrics for measuring an object-oriented software system quality including the testability, maintainability, reusability, understandability, productivity, etc
- It is time consuming to re-measure the quality of an entire system after one or some source files are modified

In the last couples of years a lot of new techniques that perform static analysis appeared. It seems that performing static analysis on the code was more important than the methods by which it was done. For example, we can do static analysis with metrics that can be defined in SQL or JAVA, with detection strategies that are defined in a descriptive language called SOD [1] and with problem detection that is based on PROLOG clauses. Because static analysis relies upon more than one language, adding a new feature is hard, for example, defining a new detection strategy implies knowing SOD and defining a new metric implies knowing SQL or JAVA.

1.2 Contribution

The contribution of this paper is to define and implement a domain specific language named, SAIL[2] that would allow a unified form of expression for the different methods of static analysis. In SAIL we'll be able to do static analysis with metrics, detection strategies and problem detection.

[1]**S**trategy **O**f **D**etection

[2]**S**tatic **A**nalysis **I**nterogative **L**anguage

1.3 Organization

The rest of this paper is organized as follows: in the next chapter(Chapter 2) we will point out the compiler's structure and additionally things about compilation techniques. After that, in Chapter 3 we give an overview of some methods by which static analysis is done, i.e. metrics, detection strategies and problem detection based on Prologue queries. As we move to Chapter 4 we present the description of the SAIL language connected to its infrastructure. Chapter 5 provides the implementation details of the SAIL interpreter. Chapter 6 provides some examples written in SAIL. The last chapter (Chapter 7) presents the conclusions of this thesis and points out to the future work of the thesis.

Chapter 2

Theoretical Foundations

This chapter offers an overview of Compilation Techniques. The first section of this chapter briefly describes the most important properties of programming languages and the second section focuses on some important characteristics of a compiler.

2.1 Properties of Programming Languages

Programming languages are often described by stating the meaning of the constructs (expressions, statements, clauses, etc.) interpretively [1]. This description implicitly defines an interpreter for an abstract machine whose machine language is the programming language. The output of the analysis task is a representation of the program to be compiled in terms of the operations and data structures of this abstract machine. By means of code generation and the run-time system, these elements are modelled by operation sequences and data structures of the computer and its basic software (operating system, etc.)

The basic of every language implementation is a language definition. Users of the language read the definitions as a user manual: What is the practical meaning of the primitive elements? How can they be meaningfully used? How can they be combined in a meaningfully way? The compiler writer, on the other hand, is interested in the question of which constructions are *permitted.*

The syntax of a language determines which character strings constitute well-formed programs in the language and which do not.

The syntactic rules of a language belong to distinct levels according to their meaning.

The lowest level contains the 'spelling rules' for basic symbols, which describe the construction of keywords, identifiers and special symbols. This rules determine, for example, whether keywords have the form of identifiers (begin) or are written with special delimiters ('BEGIN', .BEGIN), where lower case letters are permitted in addition to upper case. A common properties of these rules is that they do not affect the meaning of the program being represented.

The second level consists of the rules governing representation and interpretation of constants, for examples rules about the specifications of exponents in floating point numbers or the allowed forms of integer (decimal, hexadecimal, etc.). These rules affect the meanings of programs insofar as they specify the possibilities for direct representation of constant values. Lexical analysis treat both these syntactic rules.

The third level of syntactic rules is termed the *concrete syntax*. Concrete syntax rules describe the composition of language constructs such as expressions and statements from basic symbols. The language constructs are described by an *abstract syntax* that specifies the compositional structures of a program while leaving open some aspects of its concrete representations as a string of basic symbols.

A syntactic metalanguage is a notation for defining the syntax of a language by use a number of rules. Each rule names part of the language (called a non-terminal symbol of the language) and then defines its possible forms. A terminal symbol of the language is an atom that cannot be split into smaller components of the language.

A formal syntax definition has three distinct uses:

- it names the various syntactic parts (non-terminal symbols) of the language
- it shows which sequences of symbols are valid sentences of the language
- it shows the syntactic structure of any sentence of the language

The semantics of a language describe the meaning of a program in terms of the basic concepts of the language.

Essential semantic elements of operational languages are:

- Data objects and structures upon which operations take place
- Operations and construction rules for expressions and other operative statements
- Constructs providing flow of control, the dynamic composition of program fragments

Pragmatics relate the basic concepts of the language to concepts outside the language (to concepts of mathematics or to the objects and operations of a computer, for example).

2.2 An Overview of Compilation

2.2.1 Compilation and Interpretation

At the highest level of abstraction, the compilation and execution of a program in a high-level language look something like this:

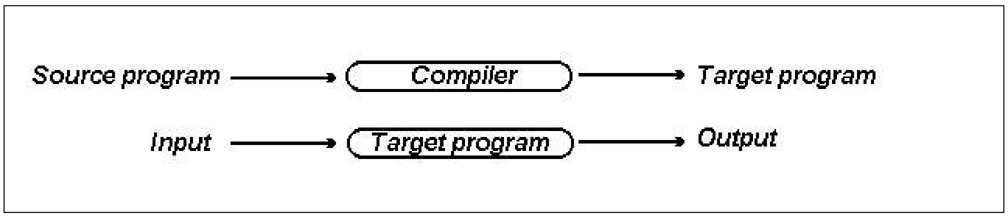

Figure 2.1: Compilation and Execution of a Program

The compiler *translates* the high-level source program into an equivalent target program (typically in a machine language). The compiler is the locus of control during compilation; the target program is the locus of control during its own execution. The compiler is itself a machine language program, presumably created by compiling some other high-level program. When written to a file in a format understood by the operating system, machine language is commonly known as *object code*.

An alternative style of implementation for high-level languages is known as

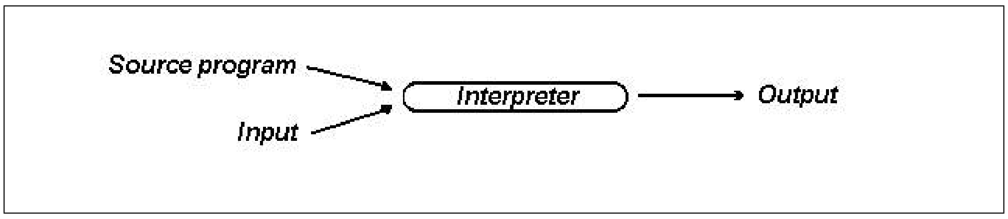

Figure 2.2: Interpretation of a Program

interpretation. Unlike a compiler, an interpreter stays around for the execution of the application. In fact, the interpreter is the locus of control during that execution. In effect, the interpreter implements a virtual machine whose "machine language" is the high-level programming language. The interpreter reads statements in that language more or less one at a time, executing them as it goes along.

Most language implementation include a mixture of both.

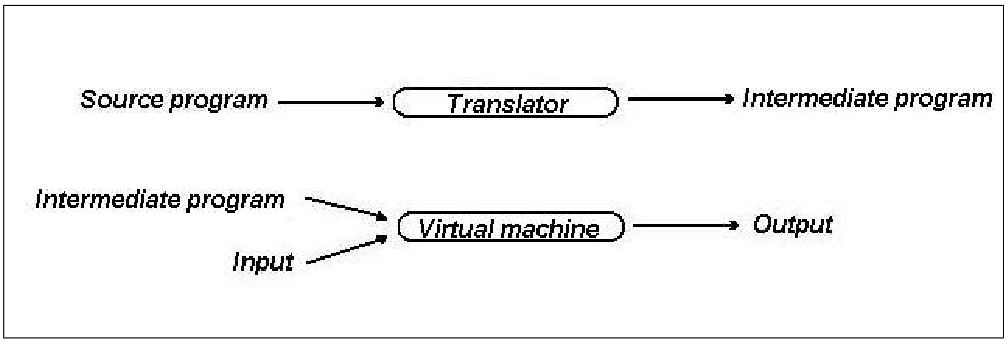

Figure 2.3: Compilation and Interpretation of a Program

2.2.2 Compiler Structure

In a typical compiler, compilation proceeds through a series of well-defined phases [2], shown in Figure 2.4. Each phase discovers information of use to later phases, or transform the program into a form that is more useful to the subsequent phase.

The first few phases (up to sematic analysis) serve to figure out the meaning

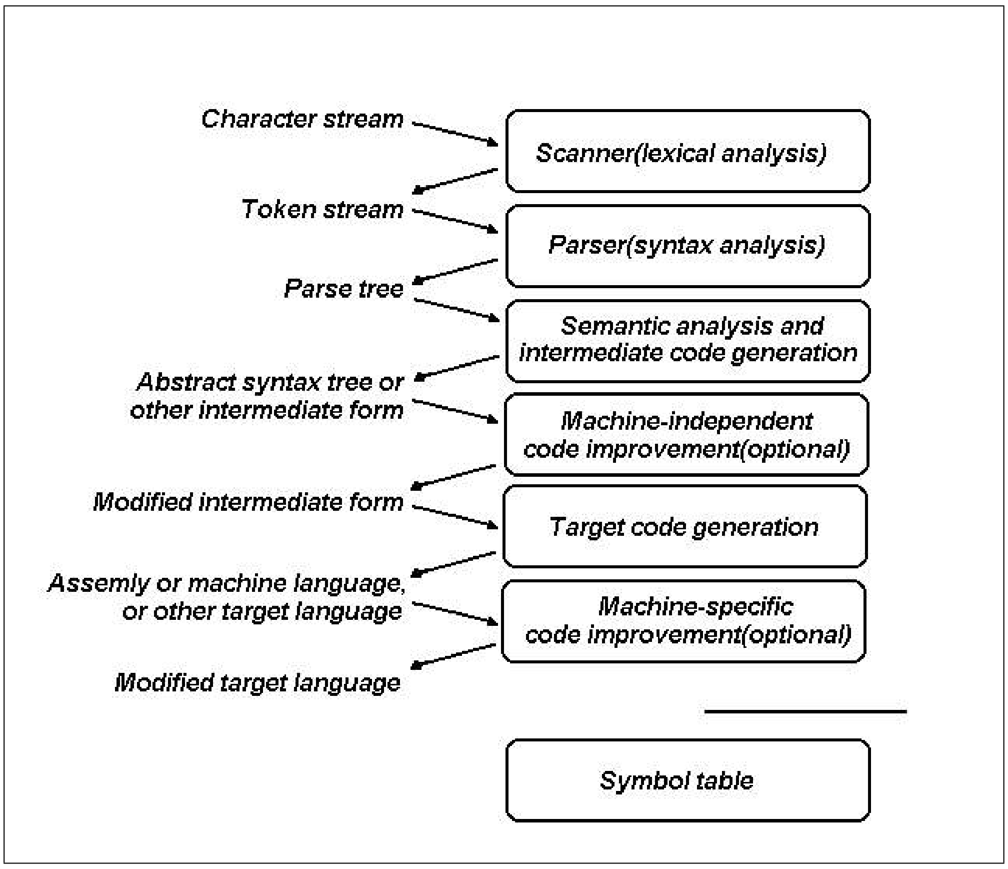
Character stream
Scanner(lexical analysis)
Token stream
Parser(syntax analysis)
Parse tree
Semantic analysis and intermediate code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement(optional)
Modified intermediate form
Target code generation
Assemly or machine language, or other target language
Machine-specific code improvement(optional)
Modified target language
Symbol table

Figure 2.4: Phases of compilation

of the source program. They are sometimes called the *front end* of the compiler. The last few phases serve to construct an equivalent target program. They are sometimes called the *back end* of the compiler. Many compiler phases can be created automatically from a formal description of the source and/or target languages.

Lexical Analysis converts the source program from a character string to a sequence of semantically-relevant symbols. The symbols and their encoding form the intermediate language output from the lexical analyzer.
In principle, lexical analysis is a subtask of parsing that could be carried out by the normal parser mechanism. To separate these functions, the source grammar G must be partitioned into subgramars G_0,G_1,G_2,... such that G_1,G_2,... describe the structure of the basic symbols and G_0 describes the structure of the language in terms of the basic symbols. L(G) is then ob-

tained by replacing the terminal symbols of G_0 by strings from LG_1,LG_2,...

The scanner also typically removes comments, produces a listing if desired, and tags tokens with line and column numbers, to make it easier to generate good diagnostics in later phases. One could design a parser to take characters instead of tokens as input - dispensing with the scanner - but the result would be awkward and slow.

Syntax Analysis of a source program determines the semantically-relevant phrases and, at the same time, verifies syntactic correctness. As a result we obtain the parse tree of the program, at first represented implicitly by the sequence of productions employed during the derivation from (or reduction to) the axiom according to the underlying grammar.

Semantic Analysis is the discovery of *meaning* in a program. The semantic analysis phase of compilation recognize when multiple occurrences of the same identifier are meant to refer to the same program entity, and ensures that the uses are consistent. In most languages the semantic analyzer tracks the *types* of both identifiers and expressions, both to verify consistent usage and to guide the generation of code in later phases.

To assist in its work, the semantic analyzer typically builds and maintains a *symbol table* data structure that maps each identifier to the information known about it. Among other things, this information includes the identifiers's type, internal structure (if any), and scope (the portion of the program in which it is valid).
Using the symbol table, the semantic analyzer enforces a large variety of rules that are not captured by the hierarchical structure of the grammar and the parse tree. For example, it checks to make sure that

- every identifier is declared before it is used
- no identifier is used in an inappropriate context (calling an integer as a subroutine, adding a string to an integer, referencing a field of the wrong type of record, etc.)
- subroutine calls provide the correct number and types of arguments
- every function contains at least one statement that specifies a return value

In many compilers, the work of the semantic analyzer takes the form of *semantic actions routines*, invoked by the parser when it realizes that it has reached a particular point within a production.
Of course, not all semantic rules can be checked at compile time. Those that can are referred to as the *static semantics* of the language. Those that must be checked at run time are referred to as the *dynamic semantics* of the language. Examples of rules that must often be checked at run time include

- variables are never used in an expression unless they have been given a value
- array subscript expressions lie within the bounds of the array
- every function specifies a value before returning

When it cannot enforce rules statically, a compiler will often produce code to perform appropriate checks at run time, aborting the program or generating an *exception* if one of the checks then fails. Some rules, unfortunately, may be unacceptably expensive or impossible to enforce, and the language implementation may simply fail to check them.

A parse tree is known sometimes as a *concrete syntax tree*, because it demonstrates, completely and concretely, how a particular sequence of tokens can be derived under the rules of the context-free grammar. Once we know that a token sequence is valid, however, much of the information in the parse tree is irrelevant to further phases of compilation. In the process of checking static semantic rules, the semantic analyzer typically transforms the parse tree into an *abstract syntax tree* by removing most of the "artificial" nodes in the tree's interior. The semantic analyzer also *annotates* the remaining nodes with useful information, such as pointers from identifiers to their symbol table entries. The annotation attached to a particular node are known as its *attributes.*
In many compilers, the annotated syntax tree constitutes the intermediate form that is passed from the front end to the back end. In other compilers, semantic analysis ends with a traversal of the tree that generates some other intermediate form. Often this alternative form resembles assembly language for an extremely simple idealized machine. In a suite of related compilers, the front ends for several languages and the back ends for several machines would share a common intermediate form.

The Code Generation phase of a compiler translates the intermediate form into the target language. Given the information contained in the syntax tree, generating correct code is usually not a difficult task (generating *good* code is harder). To generate assembly or machine language, the code generator traverses the symbol table to assign locations to variables, and then traverses the syntax tree, generating loads and stores for variable references, interspersed with appropriate arithmetic operations, tests, and branches.

Code Improvement is often referred to as *optimization*, though it seldom makes anything optimal in any absolute sense. It is an optional phase of compilation whose goals is to transform a program into a new version that computes the same result more efficiently - more quickly or using less memory, or both.
Some improvements are machine independent. These can be performed as transformations on the intermediate form. Other improvements require an understanding of the target machine (or of whatever will execute the program in the target language). These must be performed as transformations on the target program. Thus code improvement often appears as two additional phases of compilation, one immediately after semantic analysis and intermediate code generation, the other immediately after target code generation.

Symbol Table at the most basic level is a dictionary: it maps names to the information the compilers knows about them. The purpose of the symbol table is to provide unique, fixed-length encoding for the identifiers (and possibly the keywords) occurring in a program. The most basic operations which are called *insert* and *lookup* serve to place a new mapping (a name-to-object binding) into the table and to retrieve (nondestructively) the information held in the mapping for a given name.

Chapter 3

Static Analysis

This chapter presents the main contributions that exist so far in the research field of the thesis.

3.1 What is Static Analysis?

Static analysis analyze the program to obtain information that is valid for all possible executions [6].

Static source code analysis is the process by which software developers check their code for problems and inconsistencies before compiling. A static analysis might analyze the program to find all statements that potentially affect the global variables, then analyze the statements to extract information about the assigned values.

The longer a bug goes undetected the more costly it can be to identify and eradicate. A problem identified during integration can stop development and can take hundreds of developer hours to identify and correct. If a latent problem is identified by end-users it not only can cost them significant expense and dissatisfaction, it can also threaten a company's profitability. If the media identifies a latent bug in a company's new software product it could damage the company's reputation and it could threaten the ultimate success of the product.

To reduce the risks of continuing development of or shipping software with latent problems the software development organization needs to implement procedures that allow it to easily identify and eradicate problems as early as possible, preferably prior to compiling on the source code itself. While the programmer should perform code reviews, relying entirely on this manual process is time consuming, is only as reliable as the skills of the individual programmer, and will likely be conducted differently for each programmer on the team. In other words, manual pre-compile analysis can slow development, is subject to human error, and is difficult for managers to monitor and control.

By conducting static source code analysis and successfully identifying and correcting problems prior to compiling code, software developers reduce the risk and potential costs that grow exponentially in proportion to how long it takes to identify as a problem continues to go undetected.

3.2 How Is Static Analysis Done?

This section describes some methods by which Static Analysis is done.

3.2.1 Metrics

What are software metrics? Formally, they measure certain properties of a software project by mapping them to numbers (or other symbols) according to well-defined, objective measurement rules [11]. The measurement results are then used to describe, judge or predict characteristics of the software project with respect to the property that has been measured. Usually, measurements are made to provide a foundation of information upon which decisions about software engineering tasks can be both planned and performed better.

The Tight Class Cohesion Metric Bieman and Kang propose in [12] a set of two metrics on cohesion.

The TCC value is defined as the fraction of the number of methods pairs in a class, that are connected through an access to a common instance variable.

Two mechanisms by which individual methods are tied together was identified:

1. The MIV relations, that involves communication between methods through shared instance variables

2. Call relations, that involves the sending of messages directly (or indirectly) from one method to another

A client class can access only visible components of the class. In this model, invisible components of a class are included only indirectly through the visible ones. Therefore, class cohesion is modelled as the MIV relations among all visible methods (not including constructor or destructor functions) in the class.

There are three options to evaluate cohesion of a subclass:

- include all inherited components in the subclass
- include only methods and instance variables defined in the subclass
- include only inherited instance variables

Notations A method is represented as a set of instance variables directly or indirectly used by the method. The representation of a method is called abstract method AM.

- DU(M) is the set of instance variables directly used by a method M
- IU(M) is the set of instance variables indirectly used by a method M

Abstract Methods. Abstracted Class. Local Abstracted Class

Definition A class can be represented as a collection of abstract methods (AM) where each AM corresponds to a visible method in the class. The abstract method set for a method M is the union of the set of instance variables directly used by M and the set of instance variables indirectly used by M. Based on the notations we made above, the set of abstract methods for method M can be formally defined using set operators as:

$$AM(M) = DU(M) \bigcup IU(M)$$

Definition The representation of a class using abstract methods is called an abstracted class(AC). An abstracted class corresponding to a class C can be formally expressed as:

$$AC(C) = [AM(M) | M \epsilon V(C)]$$

where V(C) is the set of all visible methods in a class C and in the ancestors of C. The AM's of different methods can be identical thus there can be duplicated elements in AC. Therefore, AC is a multi-set (denoted by [and]).

Definition A local abstracted class (LAC) is a collection of AM's where each AM corresponds to a visible method defined only within the class:

$$LAC(C) = [AM(M)|M\epsilon LV(C)]$$

where LV(C) are the visible methods defined only within the class C.

Number of Direct Connections. Number of Indirect Connections The authors define two measures of class cohesion based on the direct and indirect connections of method pairs. Let NP(C) be the total number of pairs of abstract methods in AC(C). NP is the maximum possible number of direct or indirect connections in a class. If there are n methods in class C, then:

$$NP(C) = \frac{n \cdot (n-1)}{2}$$

Let NDC(C) to be the number of direct connections and NIC(C) be the number of indirect connections in AC(C).

Definition of TCC Tight class cohesion is the relative number of directly connected methods. Formally this can be expressed as:

$$NP(C) = \frac{NDC(C)}{NP(C)}$$

The implementation of the TCC metric is provided in section 6.2.

3.2.2 Detection Strategies

Detection Strategy for a design flaw is the quantifiable expression of a rule, by which design fragments, suspected of being affected by that flaw, can be found [8].

Because the problem detection approach is a metrics-based one, by "*quantifiable expression of a rule*" we mean that a rule can be properly expressed using software product metrics.

The SOD Language

In order to facilitate the definition of detection strategies, was defined a simple descriptive language, called SOD[1].In the next paragraph is presented

[1]**S**trategy **Of** **D**etection

the language as a mean by which is described how detection strategies are defined.

The Grammar

The following rules build a *simplified view* of the grammar for the SOD language. They are intended to give a better understanding of the language elements and their interaction, and *not* to describe in detail the whole grammar.

```
 DetectionStrategy    := StrategyDefinition SymbolsDefinition

# rules for definition of the Detection Strategy
StrategyDefinition    := StrategyName ":=" DetectionRule ";"
DetectionRule         := MetricWithOutliers | ComposedDetectionRule
MetricWithOutliers    := "(" MetricName "," OutlierName ")"
ComposedDetectionRule := DetectionRule CompositionOperator DetectionRule
StrategyName          := [A-z][A-z0-9_]
CompositionOperator   := "or" | "and" | "butnotin"

# symbols are definitions of metrics and outliers
SymbolsDefinition    := MetricDefinition | OutlierDefinition

# rules for the definition of the metrics
MetricDefinition := MetricName  ":=" SqlQuery ";"
MetricName       := [A-z][A-z0-9_]
 SqlQuery        := [.] # SELECT <Entity> <Value>

# rules for the definition of the outliers
OutlierDefinition := OutlierName ":="
                     OutlierType "(" OutlierParameter ")" ";"
OutlierType       := "TopValues" | "BottomValues" |
                     "HigherThan"| "LowerThan" | "BoxPlots"
OutlierName       := [A-z][A-z0-9_]
OutlierParameter  := [0-9][0-9,][%]
```

The Building Blocks of a Strategy

There are three fundamental "building stones" that collaborate to build a detection strategy:

1. *The Metrics*(lines 14-17). Metrics are used in order to express those internal characteristics of the programs that are involved in the description of a design flaw. Basically, metrics are the measurable expression of these characteristics.

2. *The Outlier Definitions*(see lines 19-24). Outlier definitions are statistical means by which the abnormal results of a measurement can be detected within a data set. They are used in order to capture the program elements (i.e. methods, classes, subsystems) that have abnormal values for a given metric. The quality of a detection strategy strongly depends on the proper selection and parameterization of an outlier definition. For the moment is working with five types of outlier-definitions (see line 21).

3. *Composition Operators*(see line 9). In a detection strategy is needed more than one metric with one outlier definition. Thus, the strategy is built as a *composition* of metrics and outlier definitions. We call the operators by which the rule is "articulated"(composed), composition operators. For the moment is using three operators: `and`, `or` and `butnotin`.

There are two possible perspectives on a detection strategy: one that focuses on the *definition* of the strategy, and the other focusing on the *results* of applying the strategy. If we focus on the definition, the composition operators are very similar to logical operators; from the results perspective a detection strategy is a composed operation using *sets* of program elements, and therefore the operators correspond to the *reunion*, *intersection* and *difference* operators from the set theory.

An Example of Detection Strategies - God Package

The Detection Rule: "God packages" are packages containing a lot of classes *and* which have many clients (classes). Therefore we used for the detection strategy two metrics: one counting the classes in a package (NOCIP) and another one counting the classes from outside the package that use the package (NOCC). For the NOCIP metric we considered as outliers all the packages having more then 20 classes. For the NOCC classes we said that the outliers are the first 20% of the packages from the system, *but* which have at least 5 client-classes.

The SOD File And now let's see how the SOD file for this detection strategy looks like:

```
# First, the detection strategy is defined as a composed rule
# involving metrics and outliers.
# Please note how a composed outlier definition is expressed
# for the NOCC metric!
GodPackage := (
       (NOCIP, NOCIP_HigherThanTwenty) and
       ((NOCC, NOCC_HighestValues) and (NOCC, NOCC_HigherThanFive))
              );

# Next, the metrics involved in the strategy are defined as SQL
#queries. The database on which the queries are executed contains
# the meta-model of the source code.

# NOCIP = Number Of Classes In Package
# NOCIP represents the number of classes contained in a package

NOCIP := SELECT f_package, count(f_class) FROM t_classes GROUP BY
f_package;

# NOCC = Number Of Client Classes

NOCC :=
SELECT f_called_package, count(f_class) AS NOCC FROM
 (SELECT DISTINCT f_called_package, f_class FROM
  ((SELECT DISTINCT f_class, f_called_package FROM t_call
    WHERE (f_package <> f_called_package)
  )
  UNION
  (SELECT DISTINCT f_class, f_provider_package FROM t_access
    WHERE (f_package <> f_provider_package)
  )) clss
) dist_clss
WHERE (f_called_package <>'')
GROUP BY f_called_package;

# Finally, the outliers used in the strategy are defined

NOCIP_HigherThanTwenty := HigherThan(20);
NOCC_HigherThanFive    := HigherThan(5);
NOCC_HighestValues     := TopValues(20%);  # top percentage
```

3.2.3 Problem Detection

In order to allow further efficient evolution of software, it has to be improved. In order to do this, we need to find out which parts of the structure prohibit further enhancements to the system and what sort of problem we are facing. We want to know which classes, subsystems or methods have to be changed and what kind of changes we must make. We call this task **problem detection** [10]. Problem detection is hard to do manually. Some of the reasons for this are:

- Programs which have to be reengineered tend to be very large.
- Systems are developed by different developers or teams
- Design problems can affect several different subsystems and thus cannot be detected locally
- It is often unclear what exactly to search for

Here we provide an example of how a design problem and a query which detects this problem may look like. We have chosen a problem related to a design guideline. If design guidelines (or heuristics) make a statement about good design, then a violation of such a guideline may indicate a design problem:

Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.

In the next paragraph is presented the formalization of the design rule described below in PROLOG:

```
% Base classes should not have knowledge about their descendants
knowsOfDerived (Class, DerivedClass) :-
  % Both Class and DerivedClass must be classes
  class (Class), class (DerivedClass),
  % DerivedClass is a direct or transitive descendant of Class
  trans (inheritsFrom, DerivedClass, Class),
  % The base class knows its heir
  knows (Class, DerivedClass).

% A class 'knows' another class, if
knows (Class1, Class2) :-
  % it inherits from that class, or
```

```
class (Class1), class (Class2),
 inheritsFrom (Class1, Class2);
 % it has an attribute of that type, or
 hasAttribute (Class1, Attr), hasType(Attr, Class2);
 % it has a method which returns an object of that type, or
 hasMethod (Class1, Meth1), returns (Meth1, Class2);
 % it has a method which calls a method of that class, or
 hasMethod (Class1, Meth1), calls (Meth1, Meth2),
   hasMethod (Class2, Meth2);
 % it has a method contain. a param. with type of that class
 hasType(Param, Class2).
```

If base classes depend on their descendants, then these descendants cannot be altered independently of their ancestors. But exactly this functionality is often needed during the evolution of an object-oriented system. So a violation of this guideline points to a spot in a software structure, where this structure is hard to change or maintain. In order to be checked automatically, this rule must now be formalized. This can be done in different ways, e.g., using different query languages. An elegant formalization of this example can be done using PROLOG. A base class would violate this heuristic, if it had knowledge of one of its direct or indirect heirs. Knowing a class means being dependent on the interface or the implementation of this class. In order to also consider the indirect heirs the transitive closure of the inheritance relation is also required. These two remarks lead us to the PROLOG clause presented below which is satisfied by the entities setting up the design problem.

3.3 Summary

In the last section was presented three ways by which static analysis is done. All of them used a meta-model for object-oriented system and all of them use a different language to query and manipulate the model. Because a dedicated language to perform static analysis doesn't exist, it is difficult to understand, improve or create methods by which it is done.

The role of SAIL is to unify the expressions for the different methods of static analysis.

Chapter 4

The SAIL Language

The goal of this chapter is to present the SAIL infrastructure connected to the meta-model extracted from the source code and to provide the language specifications.

4.1 Analysis of Source Code

The meta-model is a structured collection of design information extracted from the source code, on which the whole static analysis takes place. In this section we will first describe the tools used to extract the information and then we will briefly present the information contained in the meta-model.

4.1.1 Design Information Extractors

Some years ago appeared TableGen, that scans a C++ project and extracts the essential static design information from the source code, storing it into ASCII tables.

In the last months appeared COMPOST, an object-oriented meta-model for Java sources (called MeMoJ) which keeps the design information needed for code analysis, especially the one which is necessary to compute the metrics. A brief summary of the MeMoJ architecture is depicted in Figure 4.1.

4.1.2 The Unified Meta-Model

In order to assure that the static analysis process is language independent as much as possible, was defined a *unified format* for the data tables containing the design information from Java and C++. In other words, TableGen

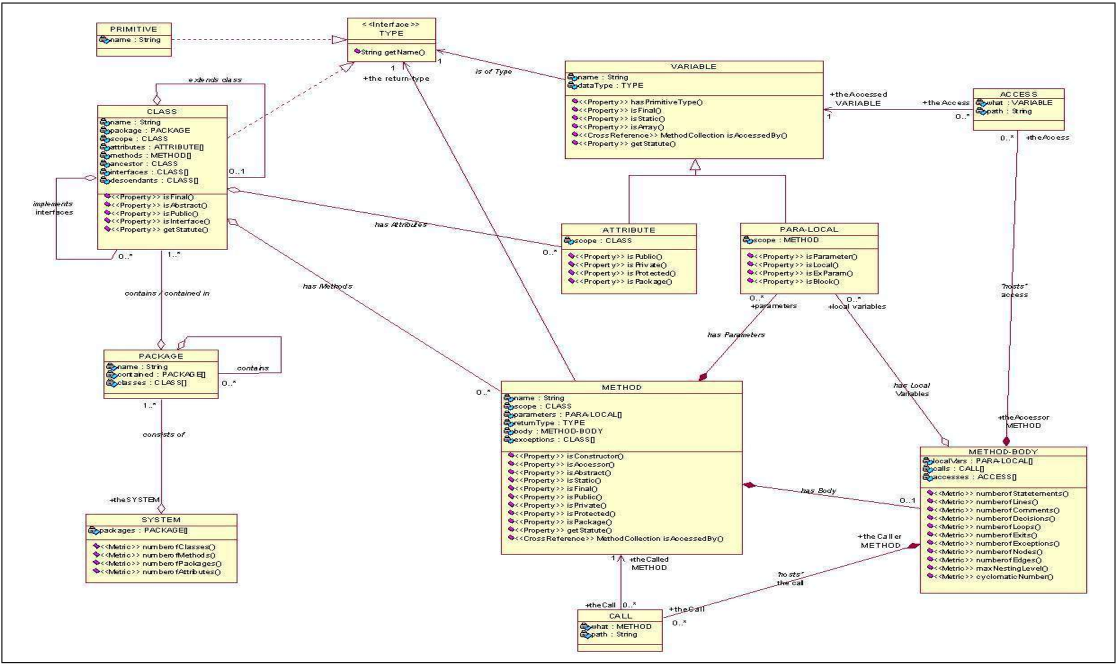

Figure 4.1: The MeMoJ Meta-model

and MeMoJ-Tables produce identical table formats. In order to interrogate the model in form of metrics, is used a relational-database server.

The *Unified Meta-Model* consists of the following data-tables:

The *Variables* Table. The reason of this table is to keep the information concerning variable declarations, i.e. global variables, class attributes, function/method parameters and local variables(including block variables). The information extracted and stored in the table can be grouped as follows:

- *Context Information.* This includes the name of the class and the signature of the function where the variable is declared.
- *Information about the Variable.* This includes the name, the base type and the complete type (which differs from the base type for pointers, references and arrays) of the variable.
- *Attributes of the Variable.* From these fields we can learn what kind of variable (e.g. global, parameter, etc.) that is and if the type is an abstract data-type.

The *Functions* Table. The goal of this table is to store any types of functions defined in the project, i.e. single functions, methods, operators, constructors and destructors. For each function, the following information is extracted:

- *Context Information.* The context consists of the name of the class where the function is defined.
- *Information about the Function.* This includes the name, the list of parameter types and the return type of the function. If a parameter has a "const" specifier this is also stored in the parameter list. If the function returns a constant value the return type is prefixed by the "const" specifier.
- *Attributes of the Function.* There are two fields in which the type of function and the special storage specifiers (virtual, static) are kept. A function can be "static", and methods can also be "virtual".

The *Classes* Table. This table contains the most important information about classes. The information included in this table can be grouped in two categories:

- *Information about the Class.* In this category we have the name of the class and the visibility scope of the class. The visibility scope for a class is not global only for *nested classes* and for classes defined within a particular *namespace.*
- *Attributes of the Class.* There are two special characteristics a class can have: it might be *abstract* or/and *generic* (template). These two boolean fields also belong to the information contained in the *Classes* table.

The *Inheritance Relations* Table. This table contains the information on *direct* inheritance relationships. This information is structured in three fields: the derived class, the parent-class and the inheritance attribute ("public", "private" or "protected").

The *Accesses of Variables* Table. The role of this table is to store all the information about the accesses (uses) of variables. This table contains second-level design information, and it relies on the first-level design information stored in the *Variables* and *Inheritance Relations* tables. We can group the information stored in this table in three categories:

- *Place of the access,* i.e. the name of the function where the access takes place, its parameters list and the class to which that function belongs;
- *The accessed variable,* i.e. the name of the variable, its base-type, the kind of variable (e.g. local, parameter), the name of the class where the variable was declared, and two boolean-fields: one indicating if the variable has a predefined or a user-defined type and the other indicating if the variables has class-scope (is declared static).

The *Invocations of Methods* Table. This table stores all the information involved in the invocation (call) of a function or a method. Similarly to the previous table, this one also contains second-level design information, and it relies on the first-level design information stored in the *Functions* and *Inheritance Relations* tables. The data contained in the table can be grouped in two categories:

- *Place of the invocation,* i.e. the name of the function where the invocation occurred,its parameters list and the class to which that function belongs.
- *Invoked function/method,* i.e. the name and the parameter list of the invoked function, the class in which the invoked method is defined, and the kind of function (e.g. "single-function", "public-method").

4.2 The SAIL Infrastructure

Figure 4.2 briefly describes the SAIL infrastructure. We'll have the following modules:

1. Initializer
   - Responsibilities
     - handles the project settings
     - instantiates the Model Implementation Adapter
     - registers the Output Modules
     - starts the SAIL Interpreter
   - Collaborations
     - Model Implementation Adapter
     - Output Formater
     - SAIL Interpreter

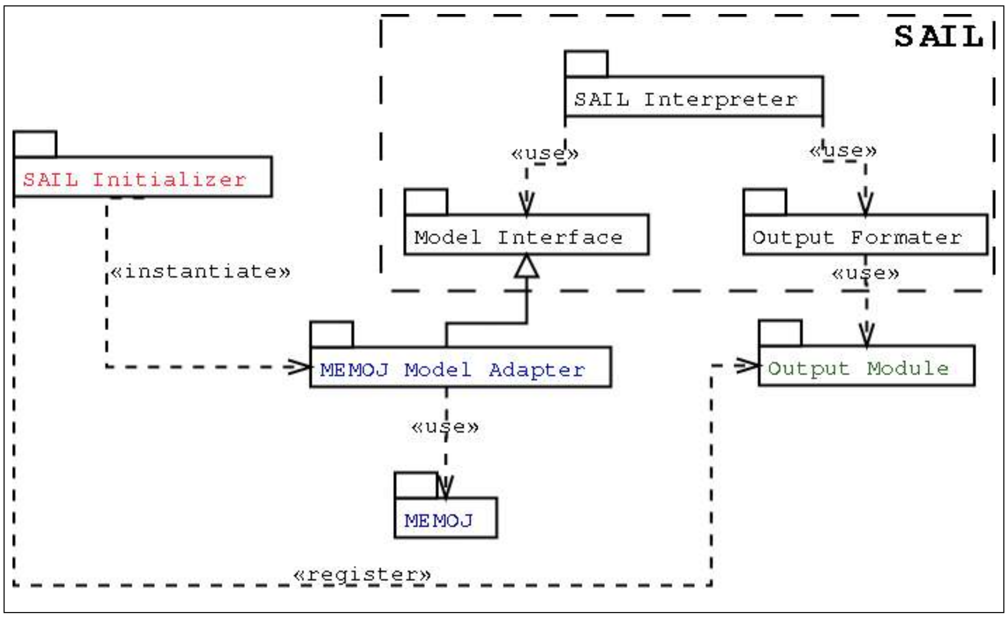

Figure 4.2: The SAIL Infrastructure

2. SAIL Interpreter
   - Responsibilities
     - interprets the input
     - sends the output results to the Output Formater
   - Collaborations
     - Model Interface
     - Output Formater
3. Model Interface
   - Responsibilities
     - provides an interface for the model structures
   - Collaborations
4. Output Formater
   - Responsibilities
     - registers the Output Modules
     - interprets the SAIL Interpreter output with the help of the Output Modules

- Collaborations
  - Output Formater Module

5. Model Implementation Adapter
   - Responsibilities
     - adapts the Model Interface to the implementation interface
   - Collaborations

6. Output Formater Module
   - Responsibilities
     - interprets the SAIL Interpreter output
   - Collaborations

4.3 Anatomy of a SAIL script

The following sections describe the language constructs of SAIL. The language is case-sensitive.
The description will be structured in a top-down manner. A SAIL script can contain the following language constructs:

- Import Statements
- Definitions of Structured-Types (Elements)
- Definitions of Functions
- Main Program

4.4 Importing SAIL Scripts

The language should provide a mechanism for reusing previously defined SAIL constructs (i.e. structured types, functions). We will do this using the import statement. The import statements, must be placed in a SAIL file before any other constructs. An import statement must will have one of the following forms:

```
import dir1.dir2.dir3.sail_file;
import dir1.dir2.*;
import sail_file;
```

In the first case, when the execution of the current SAIL file begins, the file called sail_file is loaded from the `dir1/dir2/dir3` directory. The directory path is relative to a environment variable called `SAILPATH`. In the second case, all the `SAIL` files from the specified path will be loaded. In the last case, sail_file will be loaded from the current directory, i.e. the same directory with the file where the import statement is placed.

Remark : All the SAIL files will have the `.sl` extension. In loading files through the import mechanism, the filename will not include the extension of the file. Thus in order to import a file called `sample.sl` from the same directory with the current file, you will have to write:

```
import sample;
```

4.5 Types in SAIL

In a SAIL declaration we use the following categories of types:

- **Elementary Types** – we use five elementary types: `int`, `float`, `String`, `boolean` and `void`
- **Structured Types**
- **Collections**

4.5.1 Elementary Types

Operations on Elementary Numeric Types

The operations on the int and float types are +, -, *, /, with the classical semantics found in all the languages.

String Operations Following operations should be permitted on a `string` variable:

- *concatenation*: `str1 + str2`
- *indexing*: `str[2]`
- *substring*: `str[2,5]` - gets a 4 char substring beginning with the 3rd character.
- *comparison*: `str1 == str2`; `str1 != str2`

- *wildcards*: the comparison `'alabala' == ''[*]bal[*]''` should return `true`

4.5.2 Structured Types

Structures is the name we use in SAIL for structured types. In other words in SAIL we can define new types by grouping a number of *fields* of different types.

A structured type is defined in SAIL as follows:

```
struct struct_name {
  type_name1 field_name11,field_name12,...,field_name1_n;
  type_name2 field_name21,field_name22,...,field_name2_n;
  ...
  type_name_m field_name_m_1,field_name_m_2,...,field_name_m_n; };
```

From the elementary types mentioned before, in the declaration of a structure field we can use all except the `void` type.

Example

```
  struct ClassFloatValue {
    Class clss;
    float[] values;
};
```

In the example above the structure has two fields: The first one is of a predefined structured type `Class`, that we call *model structures*(see Section 4.8); the second field is declared as a collection of float values.

Working with Structures

Defining Variables

In order to work with a structured type we will need to define a *variable* of that type. A variable should declared as seen in the next examples:

```
  ClassFloatValue aVariable;           // single declaration
  ClassFloatValue firstVar, secondVar; // multiple declaration
```

Please note that a variable declaration will be permitted only within the scope of a function or a function block. (see Section 4.6).

When a structured-type variable is created, *all* its fields are initialized as follows:

- Elementary-type fields: `int` set to 0; `float` set to 0.0; `boolean` set to `true`; `String` set to an empty string
- Collections: are initialized to an empty collection
- Structured-type fields: the same initialization rule is applied recursively on these fields

Accessing the Fields of the Structure

We will access the fields of a structure using the *dot* operator, as seen in the following example

```
Class someClassVariable;
ClassFloatValues myVar;

someClassVariable = myVar.clss;
```

In the last line of the previous example, you could see both an assignment and the use of the *dot* operator to access the `clss` member of the `myVar` variable.

4.5.3 Collections

Defining a Collection

In SAIL we will heavily use collections of structured elements. Thus, for each type we can define a collection of that type, by appending the `‘‘[]’’` suffix to the name of the type, as in the following example:

```
ClassFloatValue[] aCollection;
```

Collection Arithmetics

Following operations should be permitted between collections:

- *Union*
  – between collections of the same type (e.g. `col1 = col1 + col2`)
  – between a collection and a structure variable of the same time with the elements of the collection (e.g. `col1 = col1 + elem`)

- *Intersection*
  – between collections of the same type. The operation returns a new collection containing only the elements that are in both collections (e.g. `col3 = col1 * col2`)
  – between a collection and a structure variable returning a boolean values (e.g. `isInCollection = col1 * elem`). The semantic of the operation is to determine if the `elem` variable is in the collection or not.

- *Difference*
  – between two collections of the same type (e.g. `col3 = col1 - col2`)
  – between a collection and a structure variable (e.g. `col3 = col1 - elem`) with the following semantics: the new collection will contain all the elements from the initial collection excepting the one element that was "subtracted";

- *Cartesian Product*
  – The cartesian product between a collection of structured elements with **n** fields – called `col1` – and a second one containing elements with **m** fields – called `col2` – will create a new collection of elements that have **n + m** fields and which contains all the elements resulted from all the arrangements between the two collections. We illustrate all the mechanism involved in the cartesian product in the following example:

```
struct Struct1 {
  float firstField;
  String secondField;
};

struct Struct2 {
  int aField;
};

// please note the order of the fields
struct CartesianStruct {
  float firstField;
  String secondField;
  int thirdField;
};

...
```

```
Struct1[] col1;
Struct2[] col2;
CartesianStruct[] cart;

...

cart = (col1, col2);
```

The general syntax used for the cartesian product is:

```
(name_of_first_collection, name_of_second_collection, ...
name_of_last_collection)
```

In other words, we can compute the cartesian product not only between two collections, but between any number of collections.

- *Comparison*
  – We will use both the equal operator (`==`) and the not-equal operator (`!=`) for comparing two collections. Two collections are equal if and only if they contain the same elements. The comparison between elements is referring to their values not their physical identity.

- *Getting the number of elements from a collection*

```
int i;

// gets the number of elements from the predefined model
// collection systemClasses
i = systemClasses[];
```

4.6 Functions

4.6.1 Structure of a Function

A SAIL function has the following form, that is in fact the same with a function definition from Java or C:

```
return_type function_name(param_type1 para_name_1, ...
param_type_n param_name_n) {
  * declaration of variables
  * instructions_block
  return name_of_returned_object;
}
```

4.6.2 Parameter Passing.Return Values

The parameters are passed by value if they contain elementary values or by reference if they contain a structure or collection values.
The content of the returned (local) variable must be copied in a variable that keeps at the higher level the result of the function.

4.6.3 The Main Program

Each SAIL file will start its execution from a special function, the `main` function. This function must be uniquely defined within a SAIL file. Its syntax is slightly different from that of a usual function:

```
main(param_type1 para_name_1, ... param_type_n param_name_n) {
  * declaration of variables
  * instructions_block
}
```

The single difference so far is that the `main` function has no return type and consequently it is not ended by a `return` statement.

4.7 Statements

4.7.1 The Assignment Statement

The assignment statement is compound by a variable, an operator and an expression. The expression can be primary or compound.
Assignment is made by value when assessment of expression returns an elementary type or by reference when assessment returns a structured or a collection type.

4.7.2 The `iterate` Statement

The `iterate` statement has the following general syntax:

```
iterate(collection_name, cursor_name) {
  * declaration of variables
  * instructions_block
}
```

Obviously, both the `collection_name` and the `cursor_name` variables must be defined before their use, in the scope of the function where the iterator construct is used.

The iterator will go over all the elements from `collection_name` and do the operations specified in the instructions_block on the current element of the collection (the cursor) stored in the `cursor_name` variable.

4.7.3 The `select` Statement

A `select` statement is in fact a mechanism for "shaping" new collections from existing ones, and "filling" them with information primarily taken from the initial collection.

The general syntax of the `select` statement is:

```
select selector_construct from collector_entity
[where filter_condition]
```

Thus,there are three elements that compose a select statement: the *collector*, the *selector* and the *filter*. We will take a closer look on them.

The Collector (`collector_entity`) is the initial collection from which the information will be selected. The `collection_entity` can be a simple collection variable, or the result of any operation that results in a collection, including the result of a SAIL function that returns a collection.

Below you can see some examples on using collectors:

```
struct Person {
    int persID;
    String name;
    int age;
    String address;
}

struct Project {
    String name;
    String description;
    int persID;
}

Person[] pers, otherPers;
Project[] prj;
```

```
 String companyLabel;
 companyLabel = 'ACME';


// simple collection as collector
select name from pers where ...

// use the union operator
select * from (pers + otherPers) ...

// or define a function ...
Person[] getProjects(Person[] pc) {
   ...
}

// ... and use it as collector
select description from getProjects(pers);
```

The Selector (`selector_construct`) has the role of "shaping" (expressing) the structured-type of the result collection, i.e. to specify a number of fields, that will form together the structure of the elements that will belong to the result set. Thus, a selector is (more or less) an *enumeration of fields* enclosed in (). This can be done in one of the following manners, illustrated by the examples below:

```
// specify the field names of a collection from the
// collector_entity part of the select

select (name, address) from pers ....


// specify that you want all the fields from the collection
// resulted from the cartesian product between pers and prj

select (*) from (person, prj) ...


// you can also qualify the name of the column, by prefixing
// the name of the collection.
```

```
select person.name, prj.name from (person, prj) ...
```

In the selector we can also append some extra fields, i.e. variables of elementary or structured types that are defined in the scope of the select statement. The example below selects all the fields from the `pers` collection, but adds also a new field, `companyLabel`, that will have for all the elements the same value ('ACME')

```
select (*, companyLabel) from the pers...
```

Another possibility is to use a SAIL function in a selector. This will have the effect that the fields are those given by the result type of the function like in the following example:

```
struct ClassValue {
   Class theClass;
   int theValue;
};

ClassValue computeWMC(Class clss) { ..}

// systemClasses is the predefined collection
// containing all the classes.
// It's type is Class[]
select computeWMC(*) from systemClasses
```

The selector in the example above contains the fields of the `ClassValues` type.

The Filter (`filter_condition`) that is used to "fill" the structure shaped by the selector with a *subset* of the information found in the `collector_entity` collection.

The filter is in fact a simple or composed logical condition, of any complexity that is based on the columns of the `collector_entity` collection. The role of the filter is to take each row from the initial collection and check if it passes the "test" of the logical condition.

The filter condition may consist also of SAIL functions as long as these functions return a boolean value, as we can see in the examples below:

```
    select name from pers
    where (name != "Cristina") && (age > 22)

    // or use a SAIL function
    boolean filterFunction(Person pers) {
       return (pers.name != "Cristina") &&
              (pers.age > 22);
    }

    select name from pers
    where filterFunction(pers);
```

4.7.4 The `if/else` Statement

The branching statement has the following syntax:

```
  if(boolean_condition) {
    * declaration of variables
    * instructions_block
  }
  else {
    * declaration of variables
    * instructions_block
  }
```

The `output` Statement

The philosophy of specifying the output is the following: we want to decouple the process of formatting the output of a SAIL analysis from the analysis itself.
The `output` statement has the following syntax:

```
  output collection_variable [as entity_type]
```

where `entity_type` is limited for the moment to a reduced number of possibilities, like `metric`, `strategy` or `model` (a quality model).

4.8 The Meta-Model Information

4.8.1 The Model Structures

The SAIL language will have a number of predefined structures, called *Model structures* that reflect the main entities found in an object-oriented program, as we captured them in the MEMOJ meta-model. The structure of the model structures is described below:

The `Class` Structure

```
  struct Class {
    String  name;
    Location location;
    Class scope;

    Attribute[] attributes;
    Method[] methods;

    Class[] ancestors;
    Class[] descendants;
    Class[] interfaces;

    boolean isAbstract;
    boolean isInterface;
    boolean isPrimitive;
};
```

The `Method` and `MethodBody` Structures

```
  struct Method {
    String name;
    Location location;
    Class scope;

    Class returnType;
    ParaLocal[] parameters;
    MethodBody body;
    Class[] exceptions;

    Method[] referencedBy;
    boolean isConstructor;
    boolean isAccessor;

    boolean isAbstract;
    boolean isStatic;
    boolean isLibrary;

    boolean isPublic;
    boolean isPrivate;
    boolean isProtected;
};

  struct MethodBody {
    Location location;
    Method scope;

    ParaLocal[] localVariables;
    Method[] calls;
    Attributes[] attributeAccesses;
    ParaLocal[] variableAccesses;

    int numberOfStatemets;
    int numberOfLines;
    int numberOfComments;
    int numberOfDecisions;
    int numberOfLoops;
    int numberOfExits;
    int cyclomaticNumber;
    int maxNestingLevel;
};
```

The `Attribute`, `ParaLocal` Structures

```
  struct Attribute {
    String name;
    Class type;
    Location location;

    Method[] referencedBy;

    boolean isStatic;
    boolean isFinal;
    boolean isArray;

    Class scope;

    boolean isPublic;
    boolean isPrivate;
    boolean isProtected;
};

  struct ParaLocal {
    String name;
    Class type;
    Location location;

    Method[] referencedBy;

    boolean isStatic;
    boolean isFinal;
    boolean isArray;

    Method scope;

    boolean isParameter;
    boolean isLocal;
    boolean isGlobal;
};
```

The `Location` Structure

```
  struct Location {
    String file ;
    int startLine;
    int stopLine;
    Package package;
};
```

The `Package` Structure

```
  struct Package {
    String name;
    Class[] classes;
};
```

4.8.2 The Model Collections

In SAIL we will work on a set of predefined collections that contain all the program elements needed to analyze a system. These collections are stored and accessed as *predefined* collection variables defined as follows:

- `Class[] systemClasses`
  – all the classes in the analyzed system
- `Package[] systemPackages`
  – all the package in the system
- `Method[] systemFunctions`
  – all the methods and functions defined and/or used within the analyzed system
- `ParaLocal[] systemParaLocals`
  – all the variables (excepting attributes) that are defined in a system
- `Attribute[] systemAttributes`
  – all the attributes defined in the system

Chapter 5

The SAIL Interpreter

This chapter describes the whole design and implementation issues raised by the development of the SAIL interpreter.

5.1 Lexical and Syntax Analysis with JavaCC

5.1.1 What is JavaCC?

JavaCC is a Java parser generator written in Java [3]. It produces pure Java code. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions, debugging, etc.

Specific features of JavaCC

- TOP-DOWN JavaCC generates top-down (recursive descent) parsers as opposed to bottom-up parsers generated by YACC like tools. This allows the use of more general grammars (although left-recursion is disallowed). Top-down parsers have a bunch of other advantages (besides more general grammars) such as being easier to debug, the ability to parse to any non-terminal in the grammar, and the ability to pass values (attributes) both up and down the parse tree during parsing.
- LEXICAL AND GRAMMAR SPECIFICATIONS IN ONE FILE The lexical specifications such as regular expressions, strings, etc. and the grammar specifications (the BNF) are both written together in the same file. It makes grammars easier to read (since it is possible to use

regular expressions inline in the grammar specification) and also easier to maintain.

- TREE BUILDING PREPROCESSOR JavaCC comes with JJTree, an extremely powerful tree building preprocessor.
- DOCUMENT GENERATION JavaCC includes a tool called JJDoc that converts grammar files to documentation files (optionally in html).
- SYNTACTIC AND SEMANTIC LOOKAHEAD SPECIFICATIONS By default, JavaCC generates an LL(1) parser. However, there may be portions of the grammar that are not LL(1). JavaCC offers the capabilities of syntactic and semantic lookahead to resolve shift-shift ambiguities locally at these points. i.e., The parser is LL(k) only at such points, but remains LL(1) everywhere else for better performance. Shift-reduce and reduce-reduce conflicts are not an issue for top-down parsers.
- PERMITS EXTENDED BNF SPECIFICATIONS JavaCC allows extended BNF specifications - such as (A)*, (A)+, etc. - within the lexical and the grammar specifications. Extended BNF relieves the need for left-recursion to some extend. In fact, extended BNF is often easier to read as in A ::= y(x)* versus A ::= Ax—y.
- LEXICAL STATES AND LEXICAL ACTIONS JavaCC offers "lex" like lexical state and lexical action capabilities. Specific aspects in JavaCC that improve over other tools are the first class status it offers concepts such as TOKEN, MORE, SKIP, state changes, etc. This allows cleaner specifications as well as better error and warning messages from JavaCC.
- CASE-INSENSITIVE LEXICAL ANALYSIS Lexical specifications can define tokens not to be case sensitive either at the global level for the entire lexical specification, or on an individual lexical specification basis.
- EXTENSIVE DEBUGGING CAPABILITIES Using options DEBUG_PARSER, DEBUG_LOOKAHEAD, one can get in-depth analysis of the parsing and the token processing steps.
- SPECIAL TOKENS Tokens that are defined as special tokens in the lexical specification are ignored during parsing, but these tokens are available for processing by the tools. A useful application of this is in the processing of comments.

- VERY GOOD ERROR REPORTING JavaCC error reporting is among the best in parser generators. JavaCC generated parsers are able to clearly point out the location of parse errors with complete diagnostic information.

JJTree is a preprocessor for JavaCC that inserts parse tree building actions at various places in the JavaCC source. The output of JJTree is run through JavaCC to create the parser.

By default JJTree generates code to construct parse tree nodes for each nonterminal in the language. This behaviour can be modified so that some nonterminals do not have nodes generated, or so that a node is generated for a part of a production's expansion.

JJTree defines a Java interface Node that all parse tree nodes must implement. The interface provides methods for operations such as setting the parent of the node, and for adding children and retrieving them.

JJTree operates in one of two modes, simple and multi. In simple mode each parse tree node is of concrete type SimpleNode; in multi mode the type of the parse tree node is derived from the name of the node. If we don't provide implementations for the node classes JJTree will generate sample implementations based on SimpleNode for us. We can then modify the implementations to suit.

JJDoc takes a JavaCC parser specification and produces documentation for the BNF grammar.

Structure of JavaCC file

```
options
{
    ...
}
PARSER_BEGIN(X)
// java_compilation_unit
    ...
    X parser = new X(System.in);
    parser.first_production();
    ...
PARSER_END(X)
```

```
// the first production
void first_production() :
{...} // declarations
{...} // productions and actions
// more productions
```

Regular expressions

[“a”-“z”]	matches all lower case letters
˜[]	matches any character
˜[“\n”,“\r”]	matches any character except new line character
e1 | e2 | e3	a choice of e1, e2, e3
(e)+	one or more of occurrences of expression e
(e)*	zero or more of occurrences of expression e
[e] or (e)?	expression e is optional
((e1|e2)* [e3])|e4	nested expression

Table 5.1: Lexical Rules

5.1.2 Working with JavaCC

SAIL.jjt is the input file that JJTree will parse. In this file is described the whole SAIL language syntax and when a SAIL source will be parsed some information will be retain in the specified node of the syntax tree. For example, if we'll have some variables declarations, we'll have a SAILDeclaration node and we'll retain the type of the variables and the names of the variables. In a block we can have only a SAILDeclarationList and it has zero or more SAILDeclaration nodes.

For each product in the SAIL.jjt file ended with #Product_name JJTree will create a class with name Product_name.

SAIL.jj is the output file created by JJTree and the input file for JavaCC. JavaCC will generate the necessary classes for making Lexical and Syntax Analysis of the SAIL's programs. It is the one that will produce the SailParser class. An instance of SailParser will also makes available the syntax tree of the program written in SAIL but only if it won't discover lexical or syntactic errors in the source.

The following lines describe some products.

```
void MainFunction() #Main:
{ }
{
    "main" "(" ")" Block()
}

void Block() #Block:
{ }
{
    "{"
        DeclarationList()
        InstructionList()
    "}"
}

void InstructionList() #InstructionList:
{}
{
    ( InstructionOutput() | InstructionAssignment()
    | InstructionIterate() )*
}
```

Every program written in SAIL has a main function and that explains why we'll have a Main product that contains the predefined keyword "main",() and a Block() product.

```
void InstructionOutput() #InstructionOutput:
{
    Token t;
    String outName;
}
{
    "output" outName = Name("") ";"
    {
        jjtThis.setParameter(outName);
    }
}

String Name(String name):
```

```
{
    Token t, u;
}
{
    t = <IDENTIFIER> { name = name + t.image;}
    ( "." u = <IDENTIFIER>
    { name = name + "." + u.image ; } )*
    { return name; }
}
```

For an InstructionOutput we'll have the predefined keyword "output" and the name of the variable that we want to display and, as it is shown before, we'll retain the name of the variable in the node. Of course, the method setParameter(String param) is defined in the SAILInstructionOutput class.

```
void InstructionIterate() #InstructionIterate:
{
    String collectionName, cursorName;
}
{
    "iterate" "(" collectionName = Name("")
    { jjtThis.setCollectionName(collectionName); } ","
    cursorName = Name("")
    { jjtThis.setCursorName(cursorName); } ")"
    Block()
}
```

For an InstructionIterate we'll have the predefined keyword "iterate" followed by "(", the name of the collection that we want to iterate, the name of the cursor and ")". For this instruction we'll retain the names of the collection and cursor.

The generated files by JavaCC

- SimpleCharStream - reads the characters from the source
- Token - contains the kind of the token and a reference to the next regular token that are read from the source
- SailParserConstants - contains the constants that describe the kind of the token that are read from the source
- TokenMgrError - defines the types of lexical errors

- ParseException - defines an exception that is thrown when parse errors are encountered
- SailParser - the class that parses the source

The default generated files by JJTree

- Node - is an interface that all parse tree nodes must implement; the interface provides methods for operations such as setting the parent of the node, and for adding children and retrieving them
- SimpleNode - a default implementation of Node; by default, all the nodes of the syntax tree are SimpleNode
- SailParserTreeConstants - defines an array of string that contains the names of all the possible nodes and an array of int that contains the types of the possible nodes
- JJTSailParserState - JJTree constructs the parse tree from the bottom up; to do this it uses a stack where it pushes nodes after they have been created. When it finds a parent for them, it pops the children from the stack and adds them to the parent and finally pushes the new parent node itself. This file contains the definition of the class that provides the functions described before.

5.2 Lexical And Syntax Analysis

This steps are done by calling the method Start() defined in SailParser, class created by JavaCC. The main function defined in SailParser class will read the contents of the input file that will be interpreted. If the source code corresponds to the grammar defined for the SAIL language, the result will be the corresponding syntax tree else an exception will be thrown.

5.3 Semantic Analysis

If the source code is correct the output of the Semantic Analysis will produce a syntax tree according to the grammar of SAIL defined in **sail.jjt**.

During the Semantic Analysis all the nodes of the syntax tree are visited. If a semantic rule is violated a `SAILException` will be thrown.

All the nodes of the syntax tree are included in the javaCC package and must implement the interface *Node*, shown in Figure 5.1. It provides basic machinery for constructing the parent and child relationships between nodes.

Defined Methods in Node Interface

- `public void jjtOpen()`

  is called after the node has been made the current node; it indicates that child nodes can now be added to it

- `public void jjtClose()`

  is called after all the child nodes have been added

- `public void jjtSetParent(Node n)`

  is used for setting the parent of the node

- `public Node jjtGetParent()`

  is used for getting the parent of the node

- `public void jjtAddChild(Node n, int i)`

  tells the node to add its argument to the node's list of children

- `public Node jjtGetChild(int i)`

  returns a child node; The children are numbered from zero, left to right

- `int jjtGetNumChildren()`

  returns the number of children the node has

The class *SimpleNode* implements the Node interface. Beside the methods defined in class *Node* it has more methods.

Some Defined Methods in SimpleNode

- `public void dump(String prefix)`

  provides a rudimentary mechanism for recursively dumping the node and its children

- `public void visitTree(Visitor visitor)`

  recursively visits all nodes of the tree and for each node calls accept

- `public void interpret()`

  is used when the node is *interpreted*

- `public abstract void check(Types types) throws SailException`

  is used during the semantic analysis and is abstract because different types of nodes perform different checks; for example a *SAILDeclaration* node looks up in the symbol table to see if there is another identifier with the same name as the declaration contains

- `public void accept(Visitor v)`

  calls v.visitNode(this)

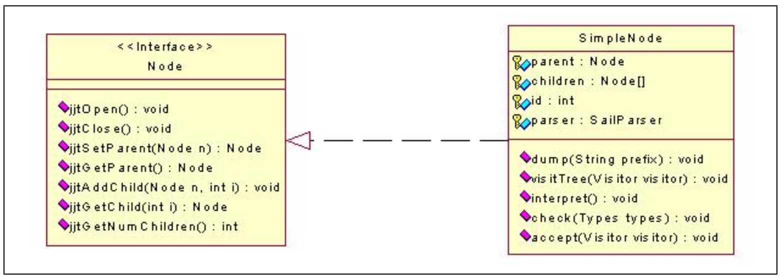
<<Interface>>
Node
jjtOpen() : void
jjtClose() : void
jjtSetParent(Node n) : Node
jjtGetParent() : Node
jjtAddChild(Node n, int i) : void
jjtGetChild(int i) : Node
jjtGetNumChildren() : int
SimpleNode
parent : Node
children : Node[]
id : int
parser : SailParser
dump(String prefix) : void
visitTree(Visitor visitor) : void
interpret() : void
check(Types types) : void
accept(Visitor visitor) : void

Figure 5.1: Node's Attributes and Operations

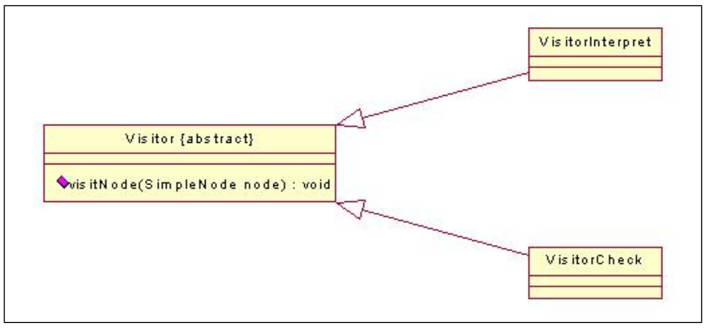

Figure 5.2: The Visitor Pattern

Design Patterns used are Composite, Singleton, Chain of Responsibility, Visitor and Template Method [4].

The Created Packages are:

- javacc - contains the generated classes by JJTree and JavaCC
- exceptions - contains the classes who's instances may be thrown during Semantic Analysis
- symbols - contains the necessarily classes in order to create types and symbol table

The following design principles was used [5]:

- The Common Closure Principle *Classes that change together, belong together.*
- The Common Reuse Principle *Classes that aren't reused together should not grouped together.*

5.3.1 Types in SAIL

All the types available in SAIL are stored in one instance of the class TypeTable. The Class Diagram is shown in Figure 5.3. Being a singleton class, it

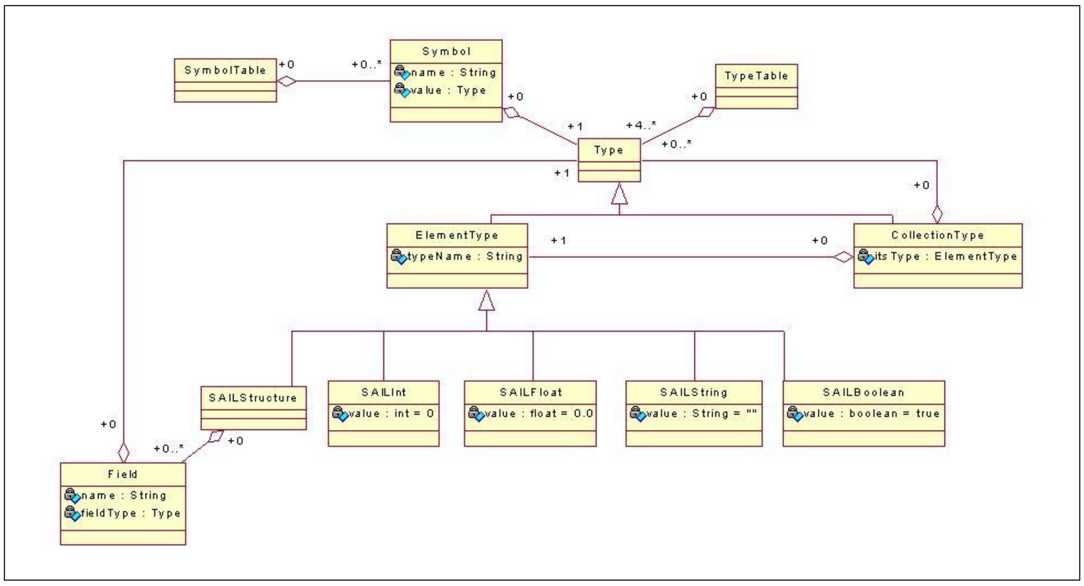

Figure 5.3: Types in SAIL

has a private constructor in which the predefined types and model structures are loaded and a static method named getInstance that guarantees only one instance is created.

```
public class TypeTable
{
  /**
   * Holds the collection of Types
   */
  private Vector tt;

  /**
   * Holds the single instance of this type
   */
  private static TypeTable instance = null;

  /**
   * Returns the single instance of this type
   */
  public static TypeTable getInstance()
  {
      if (instance == null)
          return new TypeTable();
      return instance;
  }

  /**
   * TypeTable constructor in witch the Predefined Types
   * and Model Structures are added
   */
  private TypeTable()
  {
      tt = new Vector();

      ElementType type;

      type = new SAILString();
      tt.add(type);

      ...........

      instance = this;
  }
}
```

Class CreateFieldsAndSymbols is an Abstract Factory; it has two static fields

```
public static final String ENDOFCOLLECTION = "[]";
public static final String ALLFROMCOLLECTION = "*";
```

and two static methods:

- `public static Type createElementField(SAILStructure struct, String typeOfField) throws SAILException`

  returns a type with the name typeOfField; the returned type is not a clone

- `public static Type createSymbolType(String typeOfSymbol) throws SAILException`

  returns a clone of the type named typeOfSymbol from the single instance of TypeTable if typeOfSymbol doesn't end with ENDOFCOLLECTION or a collection of type typeOfSymbol if typeOfSymbol ends with ENDOFCOLLECTION

In the following example is created a new type named Persoana corresponding to the structure declaration:

```
  struct Persoana {
    String nume;
    String prenume;
    int varsta;
};

  TypeTable tt = TypeTable.getInstance();

  SAILStructure pers = new SAILStructure("Persoana");

  try
  {
    pers.addField("nume",
    CreateFieldsAndSymbols.createElementField(pers,"String"));
    pers.addField("prenume",
    CreateFieldsAndSymbols.createElementField(pers,"String"));
    pers.addField("varsta",
    CreateFieldsAndSymbols.createElementField(pers,"int"));
    tt.add(pers);
  }
  catch (SAILException ex)
  {
    System.out.println(e.getMessage());
  }
```

5.3.2 Exceptions

Exceptions that can be thrown during Semantic Analysis are defined in the package exceptions. All are subclasses of the abstract class SAILException, subclass of Exception class defined in java.

Two design principles was used [5]:

- The Open-Closed Principle *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modifications.* The abstract class SAILException won't be modified but will be extended.
- The Dependency Inversion Principle *Depend upon Abstractions. Do not depend upon concretions.*

Possible exceptions that can be thrown are:

- SAILFieldAlreadyDefinedException
- SAILFieldNotFoundException
- SAILNoTypeForNameException
- SAILOperatorNotFitException
- SAILSelectNotPossibleException
- SAILSymbolAlreadyDefinedException
- SAILSymbolIsNotACollection
- SAILSymbolNotFoundException
- SAILTypeAlreadyDefinedException
- SAILTypesDoesNotMatchException
- SAILWrongConstantException

5.3.3 Symbol Table

Each instance of the SAILDeclarationList class has a ChainOfSymbols attribute. Here it is used the Chain of Responsibility pattern that avoids coupling the sender of a request to its receiver by giving more then one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it [4].

For the following example we'll illustrate in the Figure 5.4 how the Symbol Table looks like.

```
main()
{
  int[] collectionInt1;
  int[] collectionInt2;
  int i;
  int ii;
  int c;
  c = 3;

  iterate(collectionInt2, i)
  {
    int k;
    k = i * c;
    output k;
  }

  iterate(collectionInt1, ii)
  {
    int j;
    iterate(collectionInt2, j)
    {
      int k;
      k = i + j;
      output k;
    }
  }
}
```

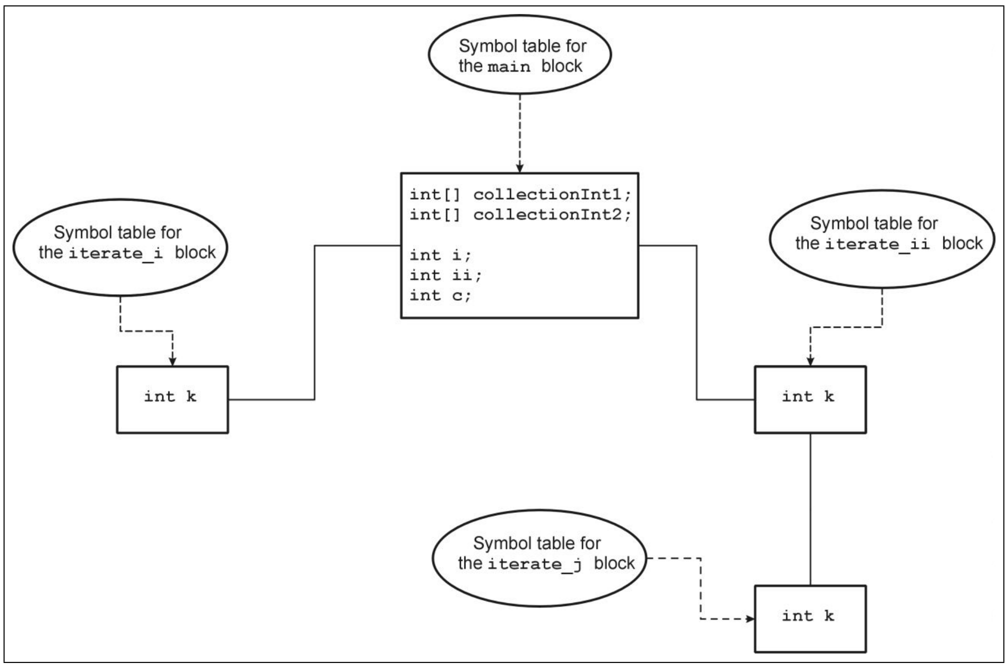
Symbol table for the main block
int[] collectionInt1;
int[] collectionInt2;
int i;
int ii;
int c;
Symbol table for the iterate_i block
int k
Symbol table for the iterate_ii block
int k
Symbol table for the iterate_j block
int k

Figure 5.4: THE SYMBOL TABLE

5.4 Interpretation of the Abstract Syntax Tree

During the interpretation phase the instructions are interpreted and the result of the code's execution is provided.

Like I mentioned before, the abstract class SIMPLENODE has the following method:

```
public void interpret()
{}
```

Every node that has to be interpreted overrides this method; only instructions are and that's way only the following classes override it:

- SAILInstructionAssignment
- SAILInstructionIterate
- SAILInstructionOutput
- SAILInstructionSelect

During the interpretation phase, for each node in the Abstract Syntax Tree we'll call the `interpret` method.

Interpretation of the `output` instruction is done by displaying the param's value, where param is the variable which we want to display. In order to do it, we'll take the symbol from the SymbolTable and we"ll print it.

```
public void interpret()
{
  System.out.print("output ");
  try
  {
    Symbol sym = ((SAILInstructionList)parent).
    getChainOfSymbolTable().getSymbol(param);
    System.out.println(sym);
  }
  catch (SAILException ex)
  {
  }
}
```

The method `getSymbol(param)` throws an exception when the symbol param is not defined; because the `interpret` method is called after the Semantic Analysis was done, in not necessary to treat the exception (Semantic analysis has already done).
Class Symbol overrides `toString` method defined in the Object class.

Chapter 6

Static Analysis with SAIL

The goal of this chapter is to prove that is possible to unify the methods by which Static Analysis is done. In order to do that, we'll comparative present the implementations of two metrics in JAVA, SQL and SAIL.

6.1 The CBO Metric

CBO = Coupling Between Objects[13]
CBO for a class is the number of other classes to which it is coupled.
Two classes are coupled when:

1. methods declared in one class use methods or
2. instance variables defined by the other class

We'll create:

- the CouplingBetweenObjects class in JAVA

```
public class CouplingBetweenObjects extends ClassMeasure {
  private HashSet used_classes;

  public CouplingBetweenObjects() {
    m_name = "CouplingBetweenObjects";
    m_fullName = "Coupling Between Objects";
  }
  ...........
}
```

- the structure `ClassValue` in SAIL

```
struct ClassValue {
    Class class;
    int value;
};
```

- the function `ClassValue computeCBO(Class crtClass)` in SAIL

```
ClassValue computeCBO(Class crtClass) {
    Method crtMethod;
    Class[] usedTypes;
    ClassValue result;

    result.class = crtClass;
    result.value = 0;
    iterate(crtClass.methods, crtMethod) {
    ...........
    }

    usedTypes = select distinct (type) from usedTypes;
    result.value = usedTypes[];

    return result;
}
```

- the main function in SAIL

```
main() {
  ClassValue[] results;
  Class crtClass;

  iterate(systemClasses, crtClass) {
    results = results + computeCBO(crtClass); }
  output results;
}
```

- a query in SQL

In order to find out the CBO value for a class, we'll have to know the number of other classes to which the class is coupled and that's why for each method defined in the class we'll count:

- the number of distinct types that the method signature uses
  - Java

```
private int para_retTypesCount(Method act_method) {
  int size, j, types_count = 0;
  ArrayList parameter_list;
  Parameter act_param;
  Type act_type;
  parameter_list = act_method.getParameterList();
  size = parameter_list.size();
  for (j = 0; j < size; j++) {
    act_param = (Parameter)parameter_list.get(j);
    act_type = act_param.getType();
    if (act_type instanceof Class &&
    !used_classes.contains(act_type)) {
      used_classes.add(act_type);
      types_count++;
    }
  }
  act_type = act_method.getReturnType();
  if (act_type instanceof Class &&
  !used_classes.contains(act_type)) {
    used_classes.add(act_type);
    types_count++;
  }
  return types_count;
  }
```

  - SAIL

```
usedTypes = usedTypes + select distinct (type)
            from crtMethod.body.attributeAccesses
            where (type != crtMethod.scope);
```

- the number of distinct types of accessed variables from the method's body
  - Java

```
private int accessTypesCount(MethodBody act_body) {
  int j, size, types_count = 0;
  ArrayList access_list = act_body.getAccessList();
  Access act_access;
  Type act_type;
  size = access_list.size();
  for (j = 0; j < size; j++) {
    act_access = (Access)access_list.get(j);
    act_type = act_access.getVariable().getType();
    if (act_type instanceof Class &&
    !used_classes.contains(act_type)) {
      used_classes.add(act_type);
      types_count++;
    }
  }
  return types_count;
}
```

  – SAIL

```
usedTypes = usedTypes + select distinct (type)
            from crtMethod.body.variableAccesses
            where (type != crtMethod.scope);
```

- the number of distinct types that are used in calls

  – Java

```
private int callTypesCount(MethodBody act_body) {
  int size, types_count = 0, j;
  ArrayList calls_list = act_body.getCallList();
  Call act_call;
  Method act_called_method;
  size = calls_list.size();
  for (j = 0; j < size; j++) {
    act_call = (Call)calls_list.get(j);
    act_called_method = act_call.getMethod();
    types_count += para_retTypesCount(act_called_method);
  }
  return types_count;
}
```

  – SAIL

```
        usedTypes = usedTypes + select distinct (scope)
                    from crtMethod.body.calls
                    where (type != crtMethod.scope);
```

The number of other classes to which the class is coupled is obtained adding the numbers described before.

- JAVA

```
public Result measure(Class act_class) {
  int size, i, count = 0;
  ArrayList method_list;
  Method act_method;
  MethodBody act_body;
  used_classes = new HashSet();

  method_list = act_class.getMethodList();
  size = method_list.size();
  for (i = 0; i < size; i++) {
    act_method = (Method)method_list.get(i);
    if (act_method.getScope() == act_class) {
      count += para_retTypesCount(act_method);
      act_body = act_method.getBody();
      if (act_body != null) {
        count += accessTypesCount(act_body);
        count += callTypesCount(act_body);
      }
    }
  }
  return new NumericalResult(act_class, count);
}
```

- SAIL

```
usedTypes = select distinct (type)
            from usedTypes;
result.value = usedTypes[];
return result;
```

- SQL

```
SELECT f_class, count(f_called_class) AS cbo FROM
(SELECT DISTINCT f_class, f_called_class FROM t_call
WHERE (f_class <> '') and f_called_class <> '')
  and (f_class <> f_called_class)
UNION
SELECT DISTINCT f_class, f_provider_class FROM t_access
WHERE (f_class <> '') and (f_provider_class <> '')
  and (f_class <> f_provider_class)) a
GROUP BY f_class
ORDER BY cbo DESC
```

6.2 The TCC Metric

TCC = Tight Class Cohesion
TCC is measuring the internal coupling of a class, i.e. the class cohesion.
Details about this metric can be found in Section 3.2.1.

We'll create:

- the structure `ClassValue` in SAIL

```
struct ClassValue {
    Class class;
    int value;
};
```

- the function `ClassValue computeTCC(Class crtClass)` in SAIL

```
ClassValue computeTCC(Class crtClass) {
  ClassValue result;
  int count = 0;
  int nrMethods = 0;
  Method[] withoutConstructors;
  Method m1, m2;
  ...........
  nrMethods = withoutConstructors[];
  result.class = crtClass;
  result.values = count / (nrMethods * (nrMethods-1));
  return result;
}
```

- the main function in SAIL

  ```
  main() {
    ClassValue[] results;
    Class crtClass;

    iterate(systemClasses, crtClass) {
      results = results +  computeTCC(crtClass);
    }
    output results;
  }
  ```

- a query in SQL

In order to find out the TCC for a class we'll:

- select from the methods defined in the class only those that aren't constructor

  ```
  withoutConstructors = select (*) from crtClass.methods
                        where (isConstructor == false);
  ```

- iterate trough the pairs of methods defined in crtClass and count the common attributes

  ```
  iterate(crtClass.methods, m1) {
    iterate(crtClass.methods, m2) {
      Attributes[] commonAttrib;
      if((m1 != m2) && (!m1.isConstructor) && (!m2.isConstructor)) {
        commonAttrib = m1.attributeAccesses * m2.attributeAccesses;
        if(commonAttrib[] != 0) {count++;}
      }
    }
  }
  ```

- perform some assigns and return the result

The corresponding implementation in SQL is:

```
SELECT all_pairs.f_class AS f_class,
      (access_pairs.cnt*100)/(all_pairs.cnt) AS tcc
FROM

  (SELECT f_class, (count(*)*(count(*)-1)/2) as cnt,
          count(*) as nr_meth
  FROM
    (SELECT DISTINCT f_class, f_function, f_signature
      FROM t_access
      WHERE (f_class <> '') and (f_function not like '%::%')
              and (f_function not like '%~%')
    ) a
   GROUP BY f_class) all_pairs,

  (SELECT x.f_class as f_class, count(x.f_class)/2 as cnt
    FROM
     (SELECT DISTINCT  a.f_class, a.f_function, a.f_signature,
                       b.f_function, b.f_signature
     FROM
      (SELECT DISTINCT f_class, f_function, f_signature,
                       f_name
      FROM t_access
      WHERE (f_use like 'attr\%') AND (f_class <> '')
            AND (f_class = f_provider_class)
            AND (f_function NOT LIKE '%::%')
            AND (f_function NOT LIKE '%~%')
      ) a,
      (SELECT DISTINCT f_class, f_function, f_signature,
                       f_name
      FROM t_access
      WHERE (f_use like 'attr\%') AND (f_class <> '')
            AND (f_class = f_provider_class)
            AND (f_function NOT LIKE '%::%')
            AND (f_function NOT LIKE '%~%')
      ) b
     WHERE (a.f_class = b.f_class)
           AND (a.f_name = b.f_name)
           AND((a.f_function <> b.f_function)
                OR (a.f_signature <> b.f_signature))
     ) x
     GROUP BY x.f_class) access_pairs
WHERE access_pairs.f_class = all_pairs.f_class;
```

6.3 Conclusions

The previous examples proved that:

- SAIL is a dedicated language in which Static Analysis can be done
- it's easier to express metrics in SAIL than JAVA or SQL
- metrics written in SAIL are easier to understand than metrics written in JAVA or SQL
- the programmer who implements metrics has to know only one programming language (SAIL)

Chapter 7

Conclusions and Future Work

7.1 Summary

The intention of this work was to analyze the possibility of unifying the methods by which Static Analysis is done.

The paper started by presenting the software's problems in nowadays and the need of Static Analysis in order to minimize them. It presented three major methods by which Static Analysis is done and pointed out some disadvantages of the existing methods and described how the apparition of SAIL can eliminate them.

SAIL language description was explained in detail.
The predefined model collections are based on the meta-model extracted from the source code. The main features of the meta-model on which the whole Static Analysis takes place are provided.

The whole implementation of SAIL was described during the following phases:

- Lexical And Syntax Analysis
- Semantic Analysis
- Interpretation of the Abstract Syntax Tree

In the last part of the paper two comparative implementations of metrics were presented with some advantages of using SAIL.

The main conclusion that results from the entire work is that Static analysis can be done just in only one programming language (SAIL). One major disadvantage is that programmers, if want to perform it, must learn another programming language (SAIL).

7.2 Future Work

1. First of all, we need experiments! We need lots of Static Analysis case-studies in order to validate the applicability and efficiency of the SAIL language.

2. At this moment the SAIL language works only the meta-model described in section 4.1.2. The Model Structures are hard-coded. We want to make possible for programmers to define their own meta-model by using the XML technology.

Appendix A

SAIL.jjt

```
options
{
  MULTI=true;
  NODE_DEFAULT_VOID=true;
}

PARSER_BEGIN(SailParser)
package javacc;

import symbols.*;
import exceptions.*;

public class SailParser
{
public static void main (String [] args)
{
  String SAIL = new String ("SailParser 1.0 (for SAIL code): ");
  SailParser parser;
  String filename = null;
  long initTime = 0;
  long parseTime = 0;
  long startTime = 0;
  long stopTime = 0;
  if (args.length != 1)
  {
    System.out.println(SAIL + "Usage is: ");
    System.out.println(" java SailParser inputfile");
    return;
  }
  else
  {
    filename = args[0];
    System.out.println(SAIL + "Reading from file " + filename);
    try
    {
      startTime = System.currentTimeMillis();
      parser = new SailParser(new java.io.FileInputStream(filename));
      stopTime = System.currentTimeMillis();
      initTime = stopTime - startTime;
    }
    catch (java.io.FileNotFoundException e)
    {
      System.out.println(SAIL + "File " + filename + " not found.");
      return;
    }
    try
    {
      Visitor visitor;
      startTime = System.currentTimeMillis();
      SAILStart n = parser.Start();
      n.dump("");

      visitor = new VisitorCheck();
      n.visitTree(visitor);

      System.out.println(TypeTable.getInstance());

      //visitor = new VisitorInterpret();
      //n.visitTree(visitor);
      n.interpretAll();

      stopTime = System.currentTimeMillis();
      parseTime = stopTime - startTime;
      System.out.println(SAIL);
      System.out.println("   SAIL program parsed " + filename +
        " successfully in " + (initTime + parseTime) + " ms.");
      System.out.println("      parser initialization time was " +
        initTime + " ms.");
      System.out.println("      parser parse time was " +
        parseTime + " ms.");
    }
    catch (ParseException e)
    {
      System.out.println(e.getMessage());
      System.out.println(SAIL + "Encountered errors during parse.");
    }
    catch (SAILException e)
    {
      System.out.println(e.getMessage());
    }
  }
}
} PARSER_END(SailParser)

/* WHITE SPACE */
SKIP : { " " | "\t" | "\n" | "\r" | "\f" }

/* COMMENTS */ MORE : { "//" : COMMENT }
<COMMENT> SPECIAL_TOKEN :
{
  <SINGLE_LINE_COMMENT: "\n" | "\r" | "\r\n" > : DEFAULT
}
<COMMENT> MORE : {  < ~[] > }

/* RESERVED WORDS AND LITERALS */
TOKEN : {
  < MAIN: "main">  | < STRING: "String" >
  | < INT: "int" > | < FLOAT: "float" >
  | < BOOLEAN: "boolean" > | < STRUCT: "struct" >
  | < OUTPUT: "output" >   | < SELECT: "select" >
  | < FROM: "from" >       | < ITERATE: "iterate">
  | < FALSE: "false" >     | < TRUE: "true" > | < NULL: "null" > }

/* LITERALS */
TOKEN :
{
  < INTEGER_LITERAL: ["1"-"9"] (["0"-"9"])* >
|
  < FLOATING_POINT_LITERAL:
  (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? (["f","F","d","D"])?
  | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
  | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
  | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"] > |
  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ > |
  < CHARACTER_LITERAL: "'" ((~["'","\\","\n","\r"])
        | ("\\" ( ["n","t","b","r","f","\\","'","\""]
        | ["0"-"7"] ( ["0"-"7"] )?
        | ["0"-"3"] ["0"-"7"] ["0"-"7"] )))"'" > |
  < STRING_LITERAL: "\"" ((~["\"","\\","\n","\r"])
        | ("\\" ( ["n","t","b","r","f","\\","'","\""]
        | ["0"-"7"] ( ["0"-"7"] )?
        | ["0"-"3"] ["0"-"7"] ["0"-"7"] )))* "\"" >
}

/* IDENTIFIERS */
TOKEN :
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* > |
  < #LETTER: ["_","a"-"z","A"-"Z"] > |
  < #DIGIT: ["0"-"9"] >
}

/* SEPARATORS */
TOKEN :
{
  <SEMICOLON: ";">| <COMMA: ",">  | <DOT: ".">
  | <LPAREN: "("> | <RPAREN: ")"> | <LBRACE: "{">
  | <RBRACE: "}"> | < LBRACKET: "[" > |
  < RBRACKET: "]" >
}

/* OPERATORS */
TOKEN :
{
  < ASSIGN: "=" > | < PLUS: "+" > |
  < MINUS: "-" >  | < STAR: "*" > | < SLASH: "/" >
}
```

```
/****************************************
 * THE SAIL LANGUAGE GRAMMAR STARTS HERE *
 ****************************************/
/*
 * Program structuring syntax follows.
 */
SAILStart Start() #Start :
{}
{
  StructureList()
  MainFunction()
  <EOF>
  { return jjtThis; }
}

void MainFunction() #Main:
{}
{
  "main" "(" ")" Block()
}

void Block() #Block:
{}
{
  "{" DeclarationList()
      InstructionList() "}"
}

void StructureList() #StructureList:
{}
{
  ( Structure(jjtThis) )*
}

void Structure(SAILStructureList node) #Structure:
{
  Token t;
}
{
  "struct" t = <IDENTIFIER> { jjtThis.setName(t.image); }
  "{" StructureDeclarationList() "}" ";"
}

void StructureDeclarationList() #StructureDeclarationList:
{}
{
  (LOOKAHEAD(2) StructureDeclaration() )*
}

void StructureDeclaration() #StructureDeclaration:
{
  Token t;
}
{
  StructureType(jjtThis) t = <IDENTIFIER>
  { jjtThis.addElement(t.image); }
  ( "," t = <IDENTIFIER>
  { jjtThis.addElement(t.image); } )* ";"
}

void StructureType(SAILStructureDeclaration jjTree):
{
  Token v=null,t,u;
  String type;
}
{
  ( type =  PrimitiveType() | u = <IDENTIFIER>
  { type = u.image; } ) [ v = "[]" ]
  {
    if (v!=null)
      jjTree.setType(type + v.image);
    else
      jjTree.setType(type);
  }
}

void DeclarationList() #DeclarationList:
{}
{
  ( LOOKAHEAD(2) Declaration(jjtThis) )*
}
```

```
void Declaration(SAILDeclarationList node) #Declaration:
{
  Token t;
}
{
  Type(jjtThis) t = <IDENTIFIER>
  { jjtThis.addElement(t.image); }
  {
    node.jjtOpen();
    node.jjtAddChildToBack(jjtThis);
    node.jjtClose();
  }
  MoreVariable(node, jjtThis.getType())
}

void MoreVariable(SAILDeclarationList node, String type):
{
  Token t;
}
{
  ("," t = <IDENTIFIER>
  {
    SAILDeclaration declaration = new SAILDeclaration
    (SailParserTreeConstants.JJTDECLARATION);
    declaration.setType(type);
    declaration.addElement(t.image);
    node.jjtOpen();
    node.jjtAddChildToBack(declaration);
    node.jjtClose();
  }
  )* ";"
}

void Type(SAILDeclaration jjTree):
{
  Token v=null,t,u;
  String type;
}
{
  ( type =  PrimitiveType() | u = <IDENTIFIER>
  { type = u.image; } ) [ v = "[]" ]
  {
    if (v!=null)
      jjTree.setType(type + v.image);
    else
      jjTree.setType(type);
  }
}

String PrimitiveType():
{
  String type = null;
  Token t;
}
{
  (  "String" {type = "String";}
  | "int" {type = "int";}
  | "float" {type = "float";}
  | "boolean" {type = "boolean";}
  )
  {
    return type;
  }
}

void InstructionList() #InstructionList:
{}
{
  ( InstructionOutput()
  | InstructionAssignment()
  | InstructionIterate() )*
}

void InstructionOutput() #InstructionOutput:
{
  Token t;
  String outName;
}
{
  "output" outName = Name("") ";"
  {
    jjtThis.setParameter(outName);
  }
}

void InstructionAssignment() #InstructionAssignment:
{
  Token t;
  String value, str;
  ElementType constant;
}
{
  str = Name("") { jjtThis.addOperand(str); }
  AssignmentOperator()
  ( ExpressionToAssign(jjtThis) | SelectToAssign(jjtThis) )
}

String Name(String name):
{
  Token t, u;
}
{
  t = <IDENTIFIER> { name = name + t.image;}
  ( "." u = <IDENTIFIER> { name = name + "." + u.image ; } )*
  { return name; }
}

ElementType Literal() :
{
  Token literal;
  ElementType constant = null;
}
{
  literal = <INTEGER_LITERAL>
  {
    try
    {
      constant = new SAILInt(literal.image);
      return constant;
    }
    catch(SAILException ex)
    {
      constant = new ElementType("int " + literal.image);
      return constant;
    }
  } |
  literal = <FLOATING_POINT_LITERAL>
  {
    try
    {
      constant = new SAILFloat(literal.image);
      return constant;
    }
    catch(SAILException ex)
    {
      constant = new ElementType("float" + literal.image);
      return constant;
    }
  } |
  literal = <STRING_LITERAL>
  {
    constant = new
    SAILString(literal.image.substring(1,literal.image.length()-1));
    return constant;
  } |
  literal = BooleanLiteral()
  {
    constant = new SAILBoolean(literal.image);
    return constant;
  }
}

Token BooleanLiteral() :
{
  Token t;
}
{
  t = "true" { return t; } |
  t = "false" { return t;}
}

Token NullLiteral() :
{
  Token t;
```

```
}
{
  t = "null"
  {
    return t;
  }
}

void AssignmentOperator() :
{}
{
  "="
}

void ExpressionToAssign(SAILInstructionAssignment instrAssig):
{
  Expression expr;
}
{
  expr = AdditiveExpression()
  { instrAssig.setExpression(expr); } ";"
}

Expression AdditiveExpression(): {
  Expression expr;
  Expression nTermExpr;
}
{
  expr = MultiplicativeExpression()
  { nTermExpr = expr; } ( ( "+" { nTermExpr = nTermExpr.addOp("+"); }
  | "-" { nTermExpr = nTermExpr.addOp("-"); })
  expr = MultiplicativeExpression() { nTermExpr.setRight(expr); } )*
  { return nTermExpr; }
}

Expression MultiplicativeExpression():
{
  Expression expr;
  Expression nTermExpr;
}
{
  expr = PrimaryExpression() { nTermExpr = expr; }
  ( ( "*" { nTermExpr = nTermExpr.addOp("*"); } | "/"
  { nTermExpr = nTermExpr.addOp("/"); } )
  expr = PrimaryExpression() { nTermExpr.setRight(expr); } )*
  { return nTermExpr; }
}

Expression PrimaryExpression():
{
  String str;
  ElementType elm;
  Expression expr = null;
}
{
  ( LOOKAHEAD(2) str = Name("")
  { expr = new TerminalExpression(str); }
  | elm = Literal() { expr = new TerminalExpression(elm); }
  | "(" expr = AdditiveExpression() ")")
  { return expr; }
}

void SelectToAssign(SAILInstructionAssignment instrAssig):
{
  SAILInstructionSelect instrSelect;
}
{
  instrSelect = InstructionSelect()
  { instrAssig.setInstructionSelect(instrSelect); }
}

SAILInstructionSelect InstructionSelect():
{
  String str;
  SAILInstructionSelect jjtThis =
  new SAILInstructionSelect(SailParserConstants.SELECT);
}
{
  "select" "(" ( "*" { jjtThis.addField("*"); }
  | str = Name("") { jjtThis.addField(str); } )+
  ("," str = Name("") { jjtThis.addField(str); } )*  ")"
  "from" str = Name("")
  { jjtThis.setCollectionSource(str); } ";"
  { return jjtThis; }
}

void InstructionIterate() #InstructionIterate:
{
  String collectionName, cursorName;
}
{
  "iterate" "(" collectionName = Name("")
  { jjtThis.setCollectionName(collectionName); } ","
  cursorName = Name("")
  { jjtThis.setCursorName(cursorName); } ")" Block()
}
```

Appendix B

BNF for SAIL

```
Start         ::= StructureList MainFunction <EOF>
MainFunction  ::= "main" "(" ")" Block
Block         ::= "{" DeclarationList InstructionList "}"
StructureList ::= ( Structure )*
Structure     ::= "struct" <IDENTIFIER>
                  "{" StructureDeclarationList "}" ";"
StructureDeclarationList ::= ( StructureDeclaration )*
StructureDeclaration     ::= StructureType <IDENTIFIER>
                             ( "," <IDENTIFIER> )* ";"
StructureType   ::= ( PrimitiveType | <IDENTIFIER> ) ( "[]" )?
DeclarationList ::= ( Declaration )*
Declaration     ::= Type <IDENTIFIER> MoreVariable
MoreVariable    ::= ( "," <IDENTIFIER> )* ";"
Type            ::= ( PrimitiveType | <IDENTIFIER> )( "[]" )?
PrimitiveType   ::= ( "String" | "int" | "float" | "boolean" )
InstructionList ::= ( InstructionOutput   |
                     InstructionAssignment | InstructionIterate )*
InstructionOutput     ::= "output" Name ";"
InstructionAssignment ::= Name AssignmentOperator
                          ( ExpressionToAssign | SelectToAssign )
Name                  ::= <IDENTIFIER> ( "." <IDENTIFIER> )*
Literal               ::= <INTEGER_LITERAL>
                          | <FLOATING_POINT_LITERAL>
                          | <STRING_LITERAL> | BooleanLiteral
BooleanLiteral        ::= "true" | "false"
NullLiteral           ::= "null"
AssignmentOperator    ::= "="
ExpressionToAssign    ::= AdditiveExpression ";"
```

```
AdditiveExpression       ::= MultiplicativeExpression
                             ( ( "+" | "-" )
                             MultiplicativeExpression )*
MultiplicativeExpression ::= PrimaryExpression
                             ( ( "*" | "/" )PrimaryExpression )*
PrimaryExpression        ::= ( Name | Literal |
                             "(" AdditiveExpression ")" )
SelectToAssign           ::= InstructionSelect
InstructionSelect        ::= "select" "(" ( "*" | Name )+
                             ( "," Name )* ")" "from" Name ";"
InstructionIterate       ::= "iterate" "(" Name "," Name ")" Block
```

Bibliography

[1] W.M. Waite, G. Goss. *Compiler Construction.* ISBN 0-387-90821-8, Springer-Verlag, Berlin, 1984.

[2] M.L. Scott. *Programming Language Pragmatics.* ISBN 1-55860-578-9, Morgan Kaufmann Publishers, 2000.

[3] *JavaCC Documentation.* http://www.webgain.com/products/java_cc, 2002.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* ISBN 0-201-63361-2, Addison-Wesley, 1997.

[5] Robert C. Martin.*Design Principles and Design Patterns.* http://www.objectmentor.com.

[6] Daniel Jackson, Martin Rinard. *Software Analysis: A Roadmap.* International Conference on Software Engineering, pp.133-145, June 2000.

[7] Radu Marinescu.*A Survey on Object-Oriented Measurement in the Context of Reengineering.* PhD Presentation, Timişoara, February 2000.

[8] Radu Marinescu.*Design Flaws and Detection Strategies.* PhD Presentation, Timişoara, June 2001.

[9] *StaticAnalysis-ProblemsAdresses.* http://www.softwareautomation.com/analysis.

[10] O. Ciupke. *Automatic Detection of Design Problems in Object-Oriented Reengineering.* Technology of Object-Oriented Languages and Systems - TOOLS 30, pages 18-32, Santa Barbara, CA, August 1999.

[11] *The* FAMOOS *Object-Oriented Reengineering Handbook.* http://www.iam.unibe.ch/famoos/handbook.

[12] J.M. Bieman, B.K. Kang. *Cohesion and Reuse in an Object-Oriented System.* Proc. ACM Symposium on Software Reusability, April 1995.

[13] S.R. Chidamber, C.F. Kemerer. *A Metrics Suite for Object-Oriented Design.* IEEE Transactions on Software Engineering, 20(6), June 1994.